Parveen Khurana

124 Followers About Following

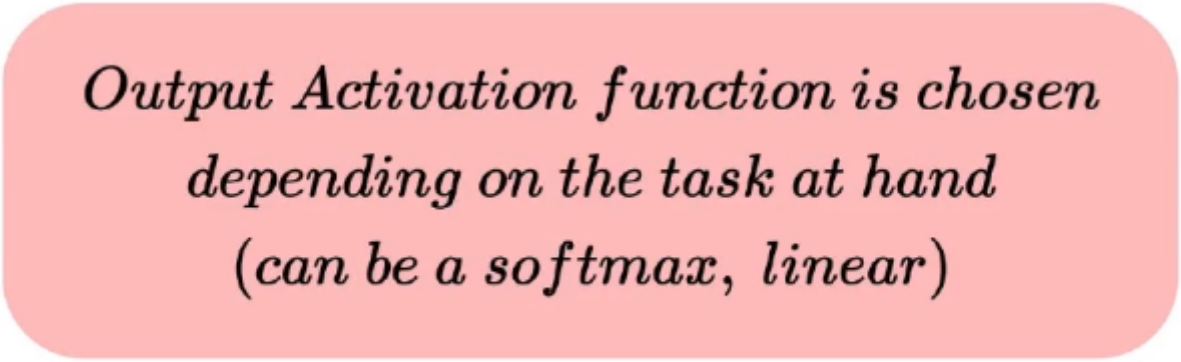
Feedforward Neural Networks — Part 2

Parveen Khurana Jan 21, 2020 · 13 min read

This article covers the content discussed in the Feedforward Neural Networks module of the Deep Learning course and all the images are taken from the same module.

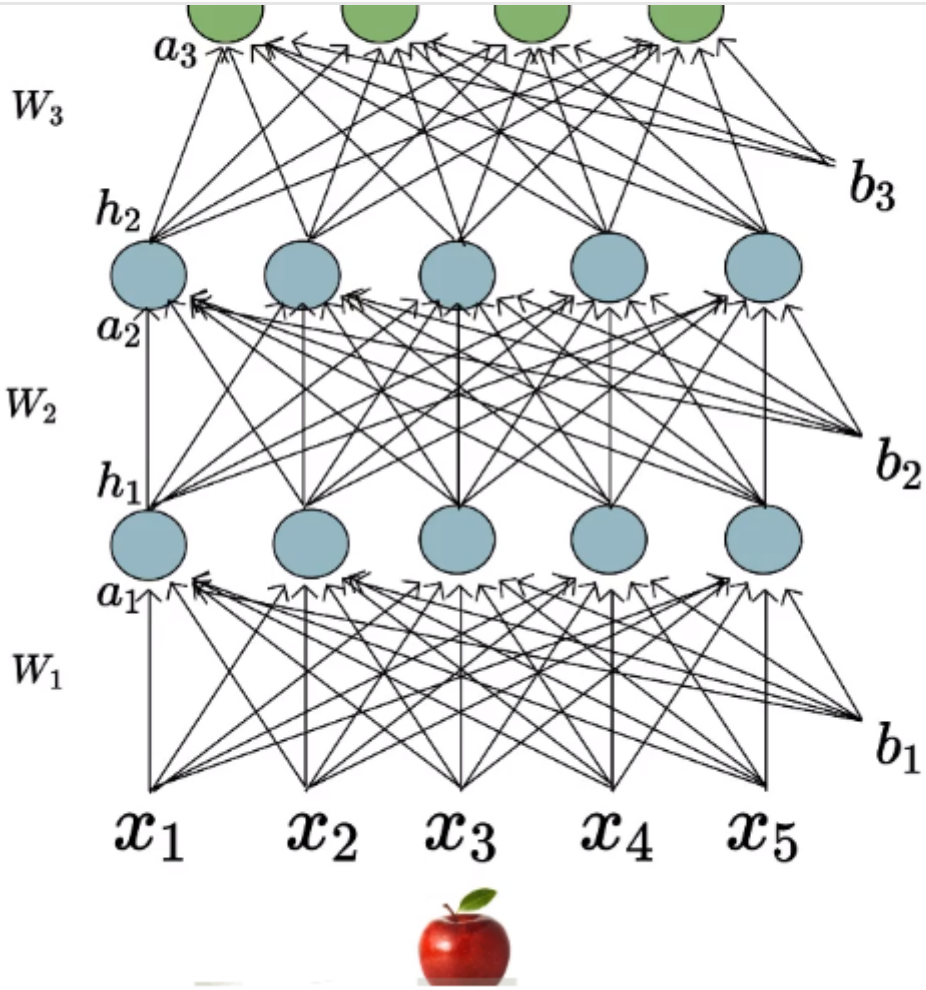
In the previous article, we discussed the Data, Tasks, Model jars of ML with respect to Feed Forward Neural Networks, we looked at how to understand the dimensions of the different weight matrix, how to compute the output. In this article, we look at how to decide the output layer of the network, how we learn the weights of the network, how to evaluate the network.

How to decide the Output Layer?



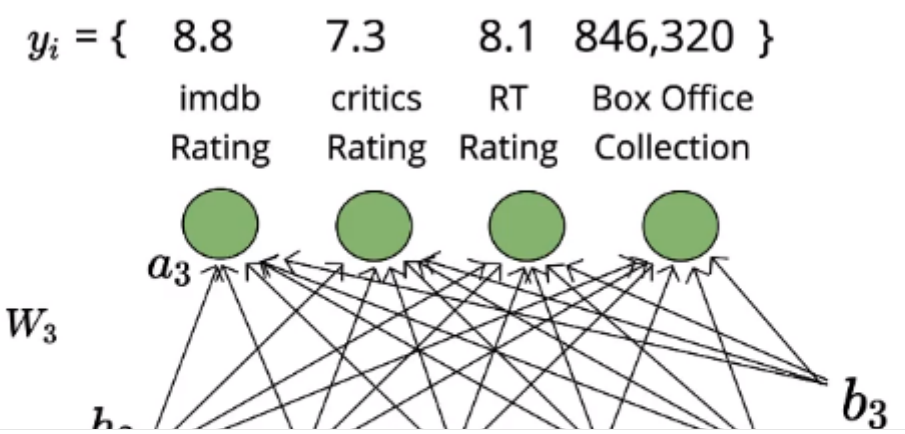
And two main tasks that we will be dealing within most of the problems are Classification and Regression:

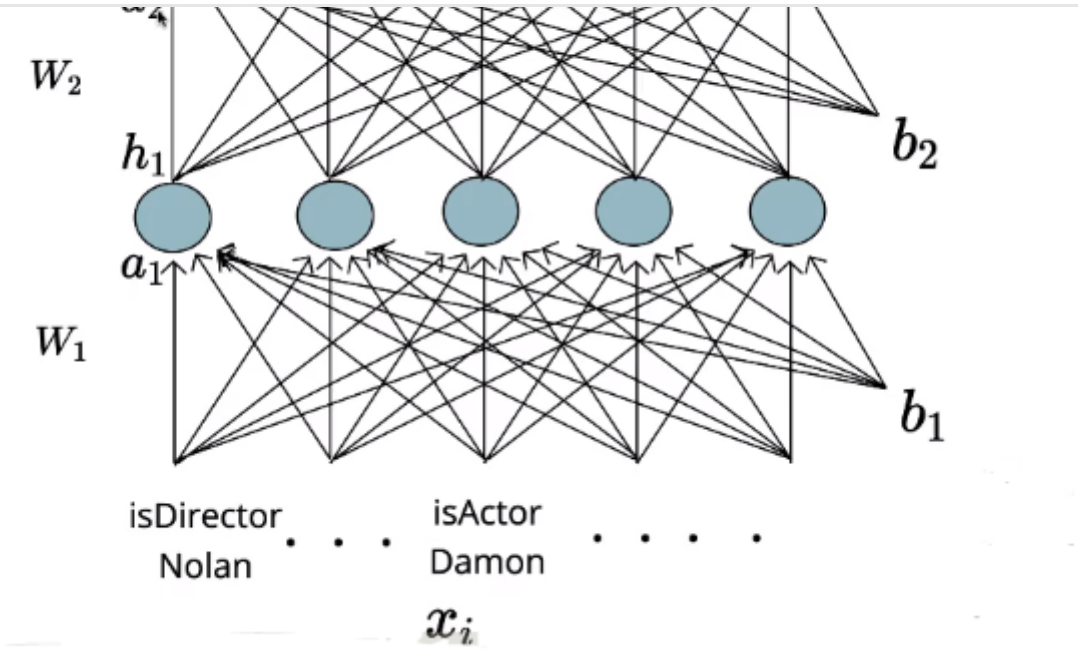
$y_i = \{ \quad 1 \qquad 0 \qquad 0 \qquad 0 \quad \}$



Classification

The above image represents the case for Multi-Class Classification. We are given an image as the input and we are passing it through all the layers and then finally predict output and we know that the true output, in this case, is a probability distribution where the entire mass is focussed on one outcome. The network will also produce a probability distribution, it will predict four values in the above case such that these 4 values sum up to 1 and each of the 4 values is greater than equal to 0.

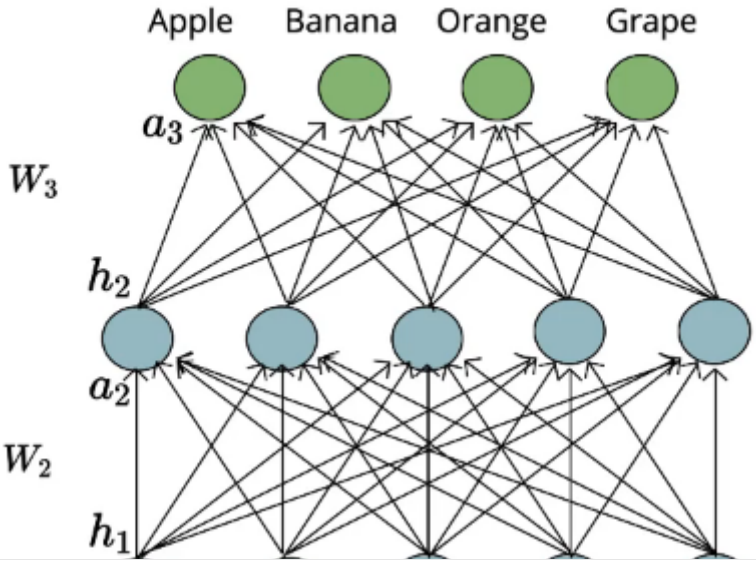


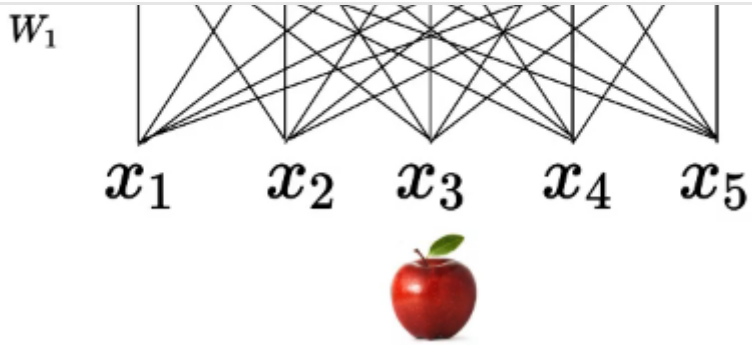


Regression

The other problem we can look at is that we are given some features of a movie and we are interested in predicting multiple metrics for example for the above case we are interested in predicting IMDB rating, critic rating, RT rating, Box Office collection. As all of these are real numbers, this is a regression problem, and in this case, also, we are trying to regress 4 values at the output. And in general, we can say that we have '**k**' outputs in the regression case where we are trying to regress '**k**' values and the output in case of Classification problem could be '**k**' values where we are predicting the probabilities corresponding to each of the '**k**' classes.

Let's see the Output Layer for the Classification problem:





True Output :

$$\hat{y} = \{\ 1,\ 0,\ 0,\ 0\ \}$$

Predicted Output :

$$\hat{y} = \{\ 0.64,\ 0.03,\ 0.26,\ 0.07\ \}$$

True output is a probability distribution and we want the predicted output also to be a probability distribution and we have passed the input up to the last layer so, in the above image of the network, we have computed till h2, so using h2 and the weights in the next layer which is W3, we can compute a3.

$$W_3 = \begin{bmatrix} w_{3\,1\,1} & w_{3\,1\,2} & . & . & . & w_{3\,1\,10} \\ w_{3\,2\,1} & w_{3\,2\,2} & . & . & . & w_{3\,2\,10} \\ w_{3\,3\,1} & w_{3\,3\,2} & . & . & . & w_{3\,3\,10} \\ w_{3\,4\,1} & w_{3\,4\,2} & . & . & . & w_{3\,4\,10} \end{bmatrix} \qquad h_2 = \begin{bmatrix} h_{2\,1} \\ h_{2\,2} \\ . \\ . \\ h_{2\,10} \end{bmatrix}$$

$$a_{3\,1} = w_{3\,1\,1} * h_{2\,1} + w_{3\,1\,2} * h_{2\,2} + w_{3\,1\,3} * h_{2\,3} + ... + w_{3\,1\,10} * h_{2\,10} + b_{31}$$

$$a_{3\,2} = w_{3\,2\,1} * h_{2\,1} + w_{3\,2\,2} * h_{2\,2} + w_{3\,2\,3} * h_{2\,3} + ... + w_{3\,2\,10} * h_{2\,10} + b_{32}$$

$$a_{3\,3} = w_{3\,3\,1} * h_{2\,1} + w_{3\,3\,2} * h_{2\,2} + w_{3\,3\,3} * h_{2\,3} + ... + w_{3\,3\,10} * h_{2\,10} + b_{33}$$

$$a_{3\,4} = w_{3\,4\,1} * h_{2\,1} + w_{3\,4\,2} * h_{2\,2} + w_{3\,4\,3} * h_{2\,3} + ... + w_{3\,4\,10} * h_{2\,10} + b_{34}$$

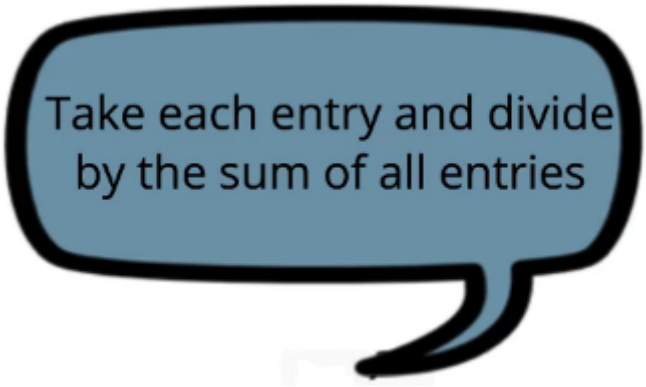
So, we have the pre-activation values for all the neurons in the output layer, in this case, we want our function to be applied to each of the 4 values that we have in a way that we get 4 probability outputs/distribution such that they sum to 1 and each of the values is greater than 0.

$$\hat{y}_1 = O(a_{31}) \qquad \hat{y}_2 = O(a_{32}) \qquad \hat{y}_3 = O(a_{33}) \qquad \hat{y}_4 = O(a_{34})$$

Let's assume for this case, we have the a3 values as:

$$Say\ a_3 = [\,3\ \ 4\ \ 10\ \ 3\,]$$

Output Activation Function has to be chosen such that output is probability



$$\hat{y}_1 = \frac{3}{(3+4+10+3)} = 0.15$$

$$\hat{y}_2 = \frac{4}{(3+4+10+3)} = 0.20$$

$$\hat{y}_3 = \frac{10}{(3+4+10+3)} = 0.50$$

$$\hat{y}_4 = \frac{3}{(3+4+10+3)} = 0.15$$

If we do the above(which is just normalization), each of the entries is going to be greater than 0 and the sum of all the values would be 1.

So, this is one way of converting the output to a probability distribution but there is a problem with this. Let's say the **a3** values for a particular case are:

$$Say\ for\ other\ input\ a_3 = [\,7 \quad -2 \quad 4 \quad 1\,]$$

a3 is computed using below equation and since all of those values are real numbers, we can get any real output, it could be positive or negative. So, in this case, probability values would look like:

$$\hat{y}_1 = \frac{7}{(7+(-2)+4+1)} = 0.70$$

$$\hat{y}_2 = \frac{-2}{(7+(-2)+4+1)} = -0.20$$

$$\hat{y}_3 = \frac{4}{(7+(-2)+4+1)} = 0.40$$

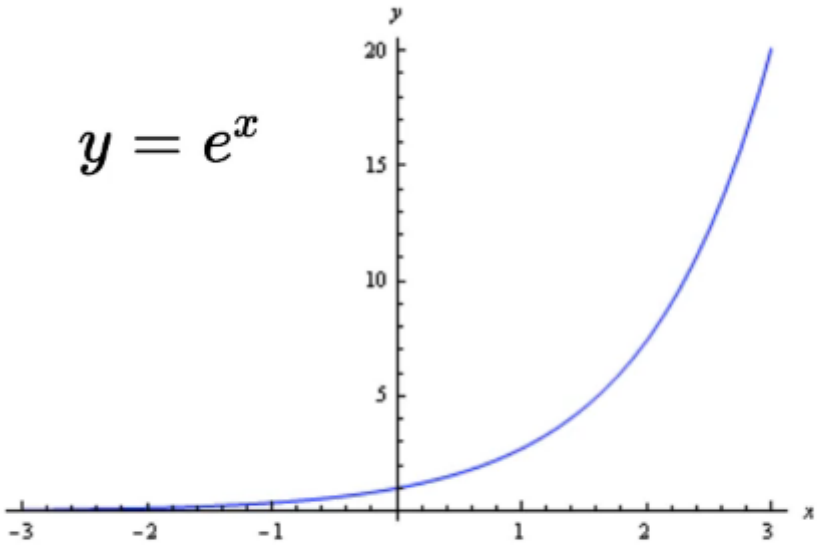
$$\hat{y}_4 = \frac{1}{(7+(-2)+4+1)} = 0.10$$

The sum of all the values is still 1 but one of the entries is negative which is not acceptable as all the values represent probabilities and by definition, the probability

Another way to convert the output into a probability distribution is to use the **softmax function**.

$$softmax(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}} \quad for\ i = 1.....k$$

The exponential function also gives back a positive output even if the input is negative.



e^x *is positive even for negative values of* x

Let's we have a vector **h**, then we can compute **softmax(h) as**:

$$h = [\ h_1 \quad h_2 \quad h_3 \quad h_4\ ]$$

$$softmax(h) = [softmax(h_1) \quad softmax(h_2) \quad softmax(h_3) \quad softmax(h_4)]$$

$$softmax(h) = \left[\frac{e^{h_1}}{\sum_{j=1}^{4} e^{h_j}} \quad \frac{e^{h_2}}{\sum_{j=1}^{4} e^{h_j}} \quad \frac{e^{h_3}}{\sum_{j=1}^{4} e^{h_j}} \quad \frac{e^{h_4}}{\sum_{j=1}^{4} e^{h_j}} \right]$$

$$softmax(h_i)\ is\ the\ i^{th}\ element\ of\ softmax\ output$$

Since we are using the exponential function in softmax, it ensures that both the numerator as well as denominators are positive for all the quantities. And also the sum of the values is still 1.

$$a_1 = W_1 * x + b_1 \qquad h_1 = g(a_1)$$

$$a_2 = W_2 * h_1 + b_2 \qquad h_2 = g(a_2)$$

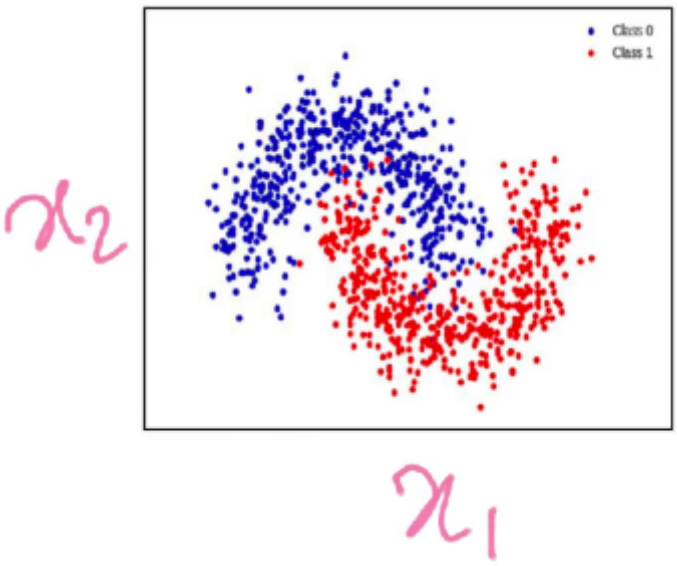
$$a_3 = W_3 * h_2 + b_3 \qquad \hat{y} = softmax(a_3)$$

So, the predicted output is going to be a probability distribution by using the Softmax function.

How to choose the right network configuration?

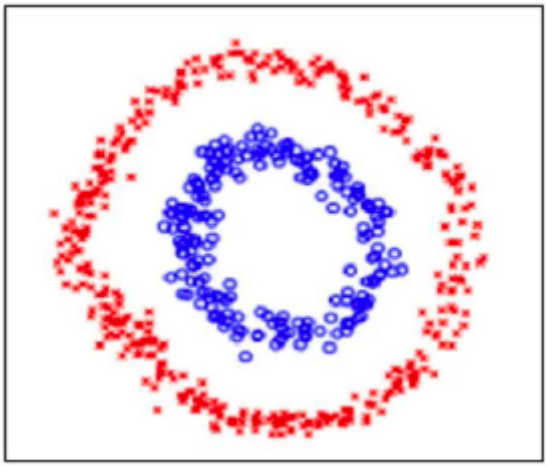
Let's say we have two input features and the final output is a function of both the inputs:

x_1	x_2
-	-

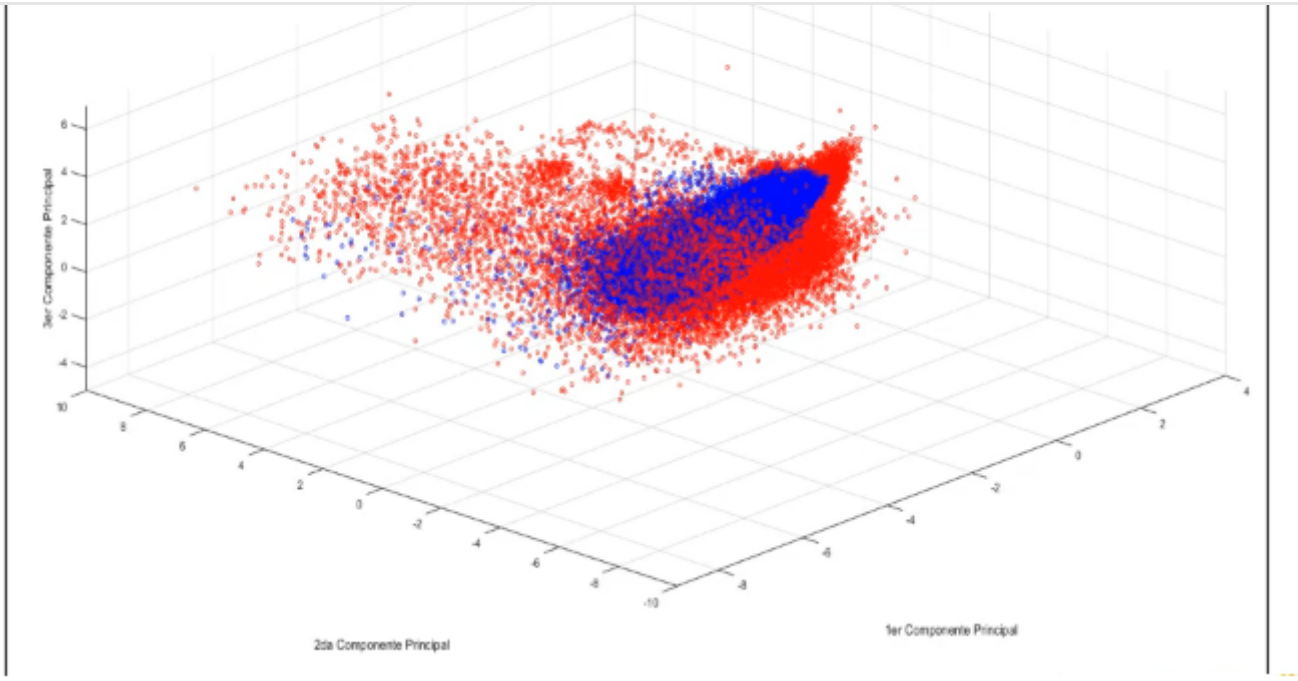


We know that we can use a DNN to deal with this kind of data. But we don't know the configuration of this DNN which will help us to deal with this data.

Another case for 2D where the data is not linearly separable is:

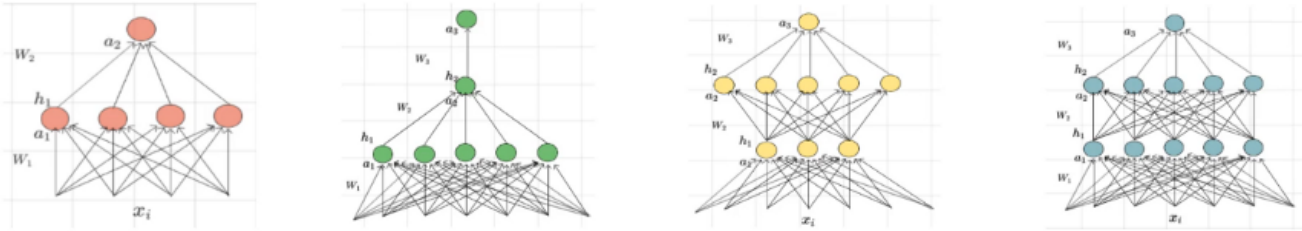


Even for 3 inputs we could plot the data and see there is some non-linearity:

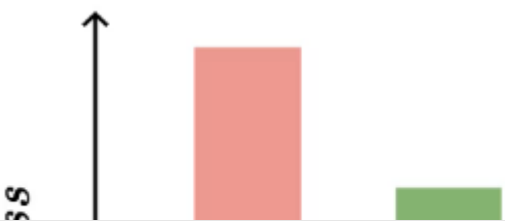


So, up to 3 dimensions we can just plot out the data and see for ourselves if the data is linearly separable or not but for higher dimensions(which is the case for most of the real-world problems), we can not just plot out the data and visualize it.

In practice, we try out different neural networks of different configurations, for example, the below image shows 4 different configurations of the neural network:



So, many configurations are possible and the way we do it in practice is that we try some of the configurations based on some intuition about how many neurons to use in each layer(which will be discussed in a different article) and then we plot the loss for each of the configurations:

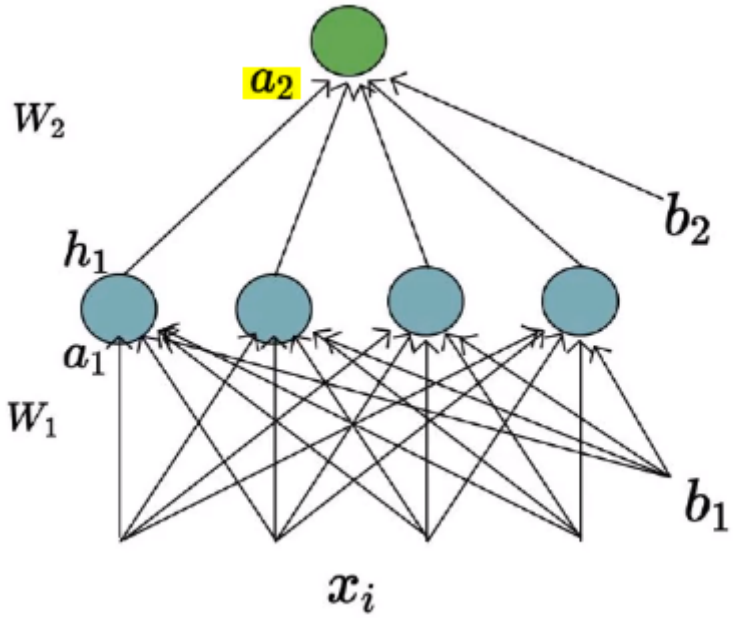


And now based on the loss value, we can select the appropriate model for the task at hand.

The loss value would depend on several hyper-parameters like the number of layers, the number of neurons in each layer, the learning rate, batch size, optimization algorithms and so on as these parameters affect the final output of the model which in turn would change the loss value. We need to check out different configurations by using different values for these parameters. This is known as Hyperparameter tuning.

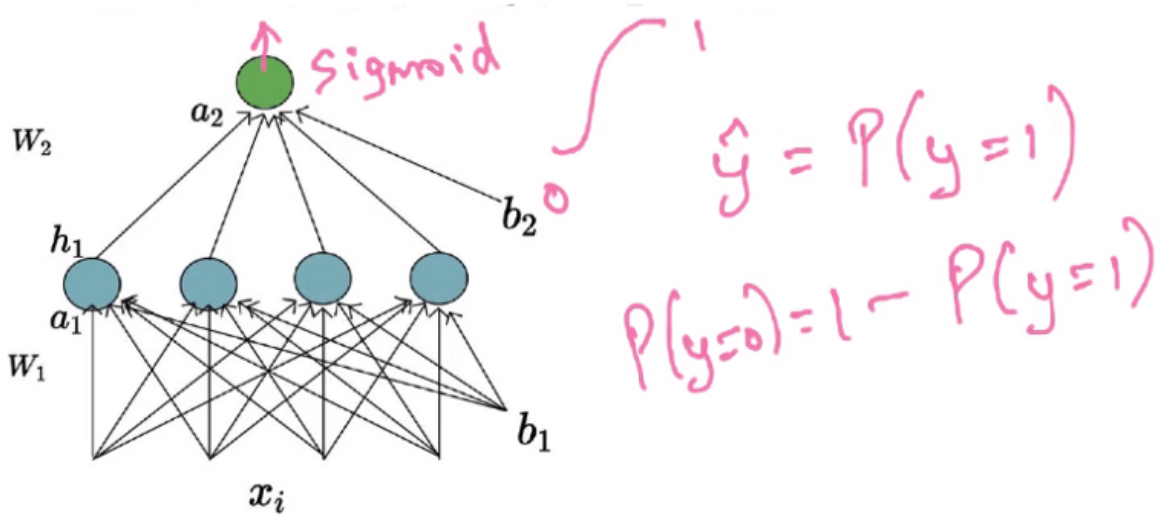
Loss Function for Binary Classification:

Our network looks like the below:



We have some inputs then the hidden layer and finally, we have this **a2**(in yellow) which aggregates all the inputs and then we have the final output.

output or we can just have one neuron in the output layer and this neuron would be a sigmoid neuron that means it is going to have a value between 0 to 1 and we can treat this value as the probability of the output being class A(say out of two classes A and B). We can also compute the probability of the output belonging to class B as both the values would have the sum as 1.



So, we can simply apply the Sigmoid(Logistics) function to **a2** and it would give the output which we can treat as the probability of belonging to class A out of two classes A and B.

Let's say the network parameters values are as below:

$$b = \begin{bmatrix} 0.5 & 0.3 \end{bmatrix}$$

$$W_1 = \begin{bmatrix} 0.9 & 0.2 & 0.4 & 0.3 \\ -0.5 & 0.4 & 0.3 & 0.3 \\ 0.1 & 0.1 & -0.1 & 0.2 \\ -0.2 & 0.5 & 0.5 & 0.7 \end{bmatrix}$$

$$W_2 = \begin{bmatrix} 0.5 & 0.8 & -0.6 & 0.3 \end{bmatrix}$$

And we want to compute the loss for a particular input(for which we already know the true output), we pass this input through all the layers of the network and compute the

$$x = [\ 0.3 \quad 0.5 \quad -0.4 \quad 0.3\ ] \qquad y = 1$$

Output :

$$a_1 = W_1 * x + b_1 = [\ 0.8 \quad 0.52 \quad 0.68 \quad 0.7\ ]$$

$$h_1 = sigmoid(a_1) = [\ 0.69 \quad 0.63 \quad 0.66 \quad 0.67\ ]$$

$$a_2 = W_2 * h_1 + b_2 = 0.948$$

$$\hat{y} = sigmoid(a_2) = 0.7207$$

As this is a classification problem, we can treat the outcome as a probability distribution and we can use the Cross-Entropy Loss to compute the loss value:

Cross Entropy Loss:

$$L(\Theta) = \begin{cases} -log(\hat{y}) & \text{if } y = 1 \\ -log(1-\hat{y}) & \text{if } y = 0 \end{cases}$$

$$L(\Theta) = -1 * \log(0.7207)$$

$$= \underline{0.327}$$

And using the loss value we can quantify how well the network is doing on the input.

Let's take another example:

$$x = [\ -0.6 \quad -0.6 \quad 0.2 \quad 0.3\ ] \qquad y = 0$$

Output :

$$a_1 = W_1 * x + b_1 = [\ 0.01 \quad 0.71 \quad 0.42 \quad 0.63\ ]$$

$$\hat{y} = sigmoid(a_2) = 0.7152$$

Cross Entropy Loss:

$$L(\Theta) = \begin{cases} -log(\hat{y}) & \text{if } y = 1 \\ -log(1-\hat{y}) & \text{if } y = 0 \end{cases}$$

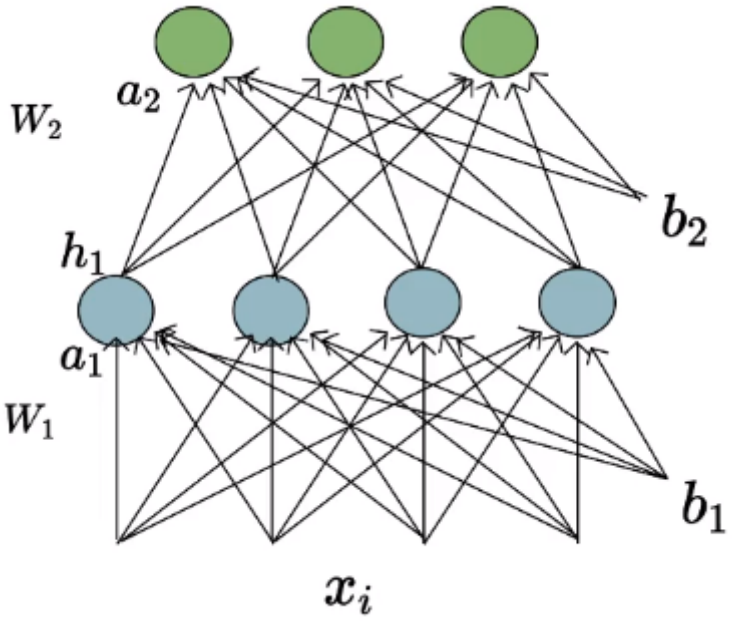
$$L(\Theta) = -1 * \log(1 - 0.7152)$$

$$= 1.2560$$

In this case, loss value is greater than that in the previous case; as in this case, the true output is 0 and the predicted is 0.7152 which is close to 1 and hence the logical explanation for why the loss is greater in this case compared to the previous example.

Loss function for multi-class classification:

Our network, in this case, would look like:



Let's say the parameters have the following value at some particular iteration:

$$b = \begin{bmatrix} 0 & 0 \end{bmatrix}$$

$$W_1 = \begin{bmatrix} 0.1 & 0.3 & 0.8 & -0.4 \\ -0.3 & -0.2 & 0.5 & 0.5 \\ -0.3 & 0.1 & 0.5 & 0.4 \\ 0.2 & 0.5 & -0.9 & 0.7 \end{bmatrix}$$

$$W_2 = \begin{bmatrix} 0.3 & 0.8 & -0.2 & -0.4 \\ 0.5 & -0.2 & -0.3 & 0.5 \\ 0.3 & 0.1 & 0.6 & 0.6 \end{bmatrix}$$

The input and true output(all the probability mass is on the second class) are as follows:

$$x = \begin{bmatrix} 0.2 & 0.5 & -0.3 & 0.3 \end{bmatrix} \qquad y = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}$$

In practice, we would be given the output as Class B, which we have contributed to the probability distribution in the above case.

We will pass the input through all the layers of the network and compute all the intermediate values as well as the final value:

Output :

$$a_1 = W_1 * x + b_1 = \begin{bmatrix} -0.19 & -0.16 & -0.09 & 0.77 \end{bmatrix}$$

$$h_1 = sigmoid(a_1) = \begin{bmatrix} 0.45 & 0.46 & 0.49 & 0.68 \end{bmatrix}$$

$$a_2 = W_2 * h_1 + b_2 = \begin{bmatrix} 0.13 & 0.33 & 0.89 \end{bmatrix}$$

$$\hat{y} = softmax(a_2) = \begin{bmatrix} 0.23 & 0.28 & 0.49 \end{bmatrix}$$

probability distribution and again, in this case, we can use the Cross-Entropy formula.

Cross Entropy Loss:

$$L(\Theta) = -\sum_{i=1}^{k} y_i \log(\hat{y}_i)$$

The probability mass on class A and class C are 0, so in effect, we have the loss value as:

Cross Entropy Loss:

$$L(\Theta) = -\sum_{i=1}^{k} y_i \log(\hat{y}_i)$$

$$L(\Theta) = -1 * \log(0.28)$$

$$= 1.2729$$

Let's consider another example:

$$x = [\ 0.6 \quad 0.4 \quad 0.6 \quad 0.1\ ] \qquad y = [\ 0 \quad 0 \quad 1\ ]$$

Output :

$$a_1 = W_1 * x + b_1 = [\ 0.62 \quad 0.09 \quad 0.2 \quad -0.15\ ]$$

$$h_1 = sigmoid(a_1) = [\ 0.65 \quad 0.52 \quad 0.55 \quad 0.46\ ]$$

$$a_2 = W_2 * h_1 + b_2 = [\ 0.32 \quad 0.29 \quad 0.85\ ]$$

$$\hat{y} = softmax(a_2) = [\ 0.2718 \quad 0.2634 \quad 0.4648\ ]$$

Cross Entropy Loss:

$$i=1$$

$$L(\Theta) = -1 * \log(0.4648)$$
$$= 0.7661$$

So, in essence, for a given input we know how to compute the forward pass or forward propagation(computing all the intermediate as well as the final value for the given input) and given the true output and the predicted output, we know how to compute the Loss value.

Given weights, we know how to compute the model's output for a given input

Given weights, we know how to compute the model's loss for a given input

So, all the above we can do on the assumption that weight values are provided to us. Let's see how we compute the weights in the Learning Algorithm.

Learning Algorithm(Non-Mathy Version):

The general recipe for learning the parameters of the model is:

Initialise w, b

Iterate over data:

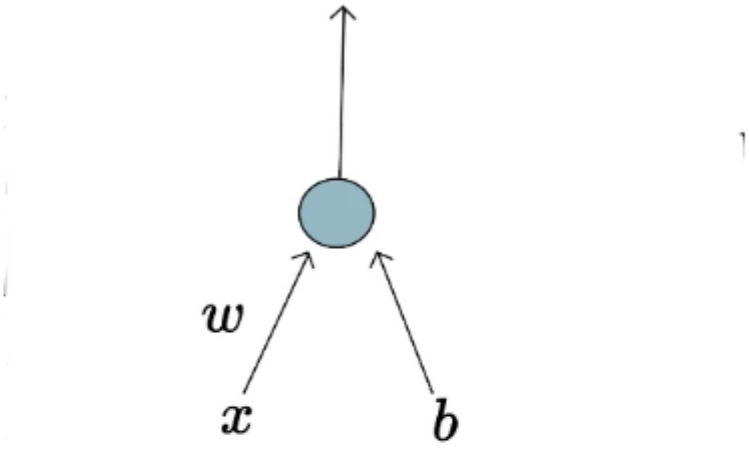
compute $\hat{y}$

$$w = w - \eta \Delta w$$

$$b = b - \eta \Delta b$$

till satisfied

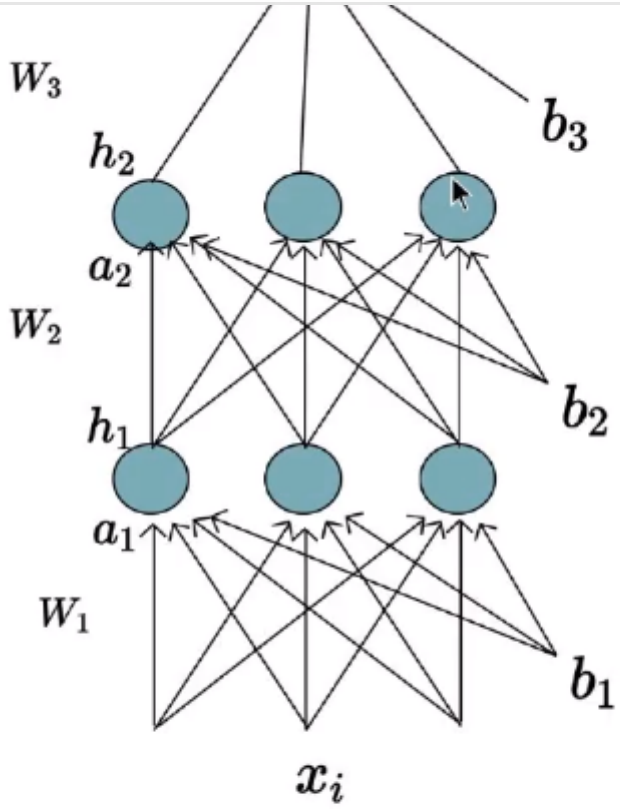
We initialize the parameters randomly, we compute the predicted output value, using which we then compute the loss value and we can feed this loss value to the frameworks like PyTorch or TensorFlow, and the framework will give us back the delta value of the change that is required.



$$Earlier : w, b$$

$$Now : w_{111}, w_{112}, \ldots$$

Earlier we had just two weights **w, b**. Now we have many more weights and biases, all the weights across all the layers. And earlier the loss function was dependent on **w,b** but **in the case of a DNN, the loss function depends on all the parameters(all the weights, all the biases)**.



$$Earlier : L(w, b)$$

$$Now : L(w_{111}, w_{112}, ...)$$

And we can use the same recipe for the Learning Algorithm in the case of a DNN as in case of Sigmoid Neuron:

Initialise w, b

Iterate over data:

$compute\ \hat{y}$

$compute\ \mathscr{L}(w, b)$

$$w_{111} = w_{111} - \eta \Delta w_{111}$$

$$w_{112} = w_{112} - \eta \Delta w_{112}$$

$$w_{313} = w_{313} - \eta \Delta w_{313}$$

till satisfied

We initialize all the weights in the network and all the biases with some random value, now using this parameter configuration, we are going to iterate over all the training data that is given to us, we compute the final output for all the training inputs. Once we have the predicted outputs, we can compute the loss value using the Cross-Entropy formula. And once we have the loss value, we can feed it to frameworks that would then tell us how to update all of the parameters appropriately in a way such that the overall loss decreases. These parameters are just going to be the partial derivative of the loss function with respect to each parameter and the way to compute this is discussed in another article.

Evaluation:

We are given some data with true labels and we can pass each data point through the model and compute the predicted output for each data point.

Indian Liver Patient Records * - whether person needs to be diagnosed or not ?

Test Data

Age	Albumin	T_Bilirubin		y	Predicted
65	3.3	0.7		0	0
62	3.2	10.9	. . .	0	1
20	4	1.1		1	1
84	3.2	0.7		1	0

Indian Liver Patient Records * - whether person needs to be diagnosed or not ?

Test Data

Age	Albumin	T_Bilirubin	y	Predicted
65	3.3	0.7	0	0

84	3.2	0.7	1	0

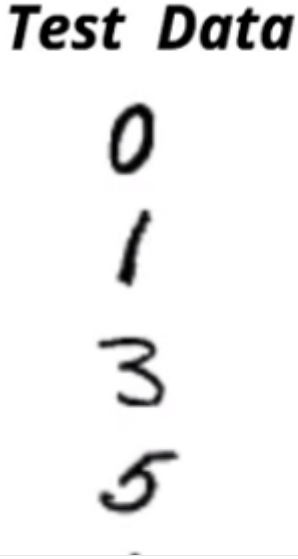
We can compute the Accuracy as:

$$Accuracy = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

$$= \frac{2}{4} = 50\%$$

The only thing to note is that how are we going to get the predicted output as 0 or 1. In the case of Binary Classification, where we have the sigmoid neuron at the output layer, we can have some threshold value to decide this for example. let say sigmoid's output is 0.6 and we can have threshold as 0.5 meaning anything greater than 0.5 would be treated as 1 and anything less than equal to 0.5 would be considered as 0. Another way of looking at this would be: say sigmoid's output is 0.6 which can be written as the probability distribution: [0.4 0.6], so the label that we are going to assign as the predicted output is the index of the maximum value in the probability distribution.

And the same thing is applicable for Multi-class classification also:



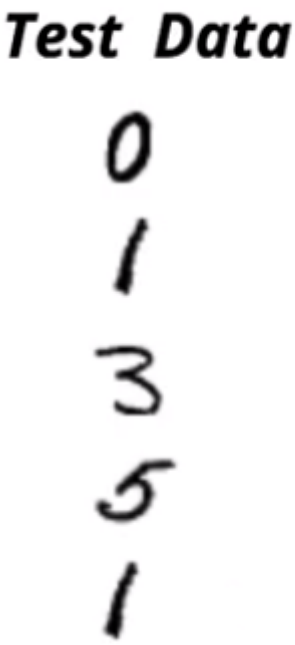
y	Predicted
0	0
1	7
3	8
5	5
1	1

$$Accuracy = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

$$= \frac{3}{5} = 60\%$$

In the case of Multi-class classification, we could also look for per-class accuracy meaning to say for example: of all the 1's images that we have in the test set, how did the model do on that, so we check the output for those images and compute the accuracy using the same formula as in the above image.

For example: for the case of 1's images, we have a total of 2 images out of which one has been classified correctly as 1 giving a total accuracy of 1/2 i.e 50%.



y	Predicted
0	0
1	7
3	8
5	5
1	1

In this way, we can compute the class-wise accuracy for all the classes and that helps in analysis sometimes for ex. for the above case, overall accuracy is 60% but for 1's images, the accuracy was 50%. Let's say for class 9 it is 80%; so on average it is 60% but there are some classes on which we the model is doing pretty well but for some

Summary:

Data: Inputs are going to be real numbers.

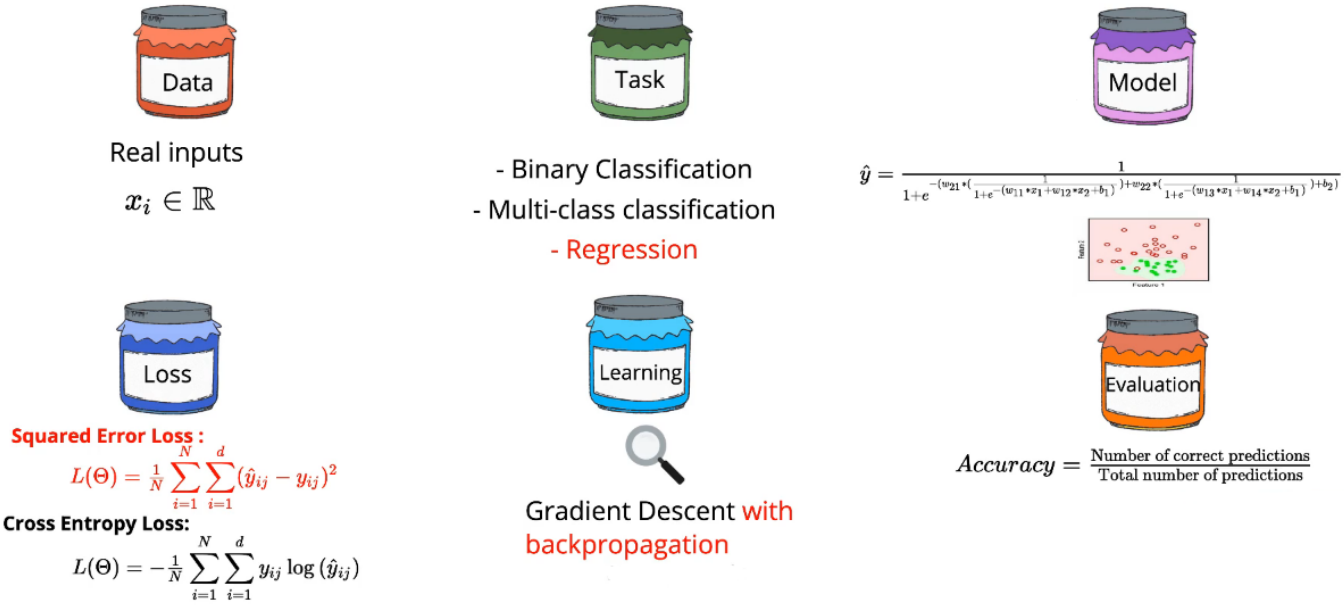
Task: We are dealing with Binary Classification, Multi-Class Classification, Regression tasks using a DNN of which the Classification part is covered in this article.

Model: We are dealing with non-linear model and we can now deal with not linearly separable data.

Loss function: We use the Cross-Entropy loss function for the Classification task, we compute the cross-entropy loss for each data point and then just average it.

Learning Algorithm: the same recipe of Gradient Descent is used for training a DNN.

Evaluation: We use the Accuracy metric. Sometimes it makes sense to compute per class accuracy in case of Multi-class classification tasks.



Artificial Intelligence Deep Learning Feed Forward Neural Network Algorithm Data Science